python

Dictionaries

Python Dictionary

- The dictionary itself is an abstract datatype.
- In this it contains a group of data with the different data types.
- And each data is stored as a key-value pair, that key is used to identify that data.
- We can add or replace values of the dictionary, it is mutable.
- But the **key associated** with the dictionary can't be changed.
- Therefore, the keys are immutable or unique.

The compound types —strings, lists, and tuples—use integers as indices. If you try to use any other type as an index, you get an error.

Dictionaries are similar to other compound types except that they can use any immutable type as an index.

One way to create a dictionary is to start with the empty dictionary and add elements. The empty dictionary is

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
```

```
#WORK ON Dictionary
x={ }
x[1] ="DR.Ramandeep"
x['2'] ="DR.Sandhu"
x['3'] ="Like Python"
print(x)
```

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
```

The first assignment creates a dictionary named **eng2sp**; the other assignments add new elements to the dictionary. We can print the current value of the dictionary in the usual way:

```
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```

The elements of a dictionary appear in a comma-separated list. Each entry contains an index and a value separated by a colon. In a dictionary, the indices are called **keys**, so the elements are called **key-value pairs**.

```
#WORK ON Dictionary
x={ }
x[1] ="DR.Ramandeep"
x['2'] ="DR.Sandhu"
x['3'] ="Like Python"
print(x)
```

```
{1: 'DR.Ramandeep', '2': 'DR.Sandhu', '3': 'Like Python'}
```

Dictionary operations

The `del` statement removes a key-value pair from a dictionary. For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
'pears': 217}
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
```

If someone buys all of the pears, we can remove the entry from the dictionary:

```
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
```

Or if we're expecting more pears soon, we might just change the value associated with pears:

```
>>> inventory['pears'] = 0
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
```

The `len` function also works on dictionaries; it returns the number of key-value pairs:

```
>>> len(inventory)
4
```

Dictionary methods

A **method** is similar to a function—it takes arguments and returns a value—but the syntax is different. For example, the `keys` method takes a dictionary and returns a list of the keys that appear, but instead of the function syntax `keys(eng2sp)`, we use the method syntax `eng2sp.keys()`.

keys( )
values()
items()

```
>>> eng2sp.keys()
['one', 'three', 'two']
```

This form of dot notation specifies the name of the function, `keys`, and the name of the object to apply the function to, `eng2sp`. The parentheses indicate that this method has no parameters.

A method call is called an **invocation**; in this case, we would say that we are invoking `keys` on the object `eng2sp`.

The values method is similar; it returns a list of the values in the dictionary:

```
>>> eng2sp.values()
['uno', 'tres', 'dos']
```

The items method returns both, in the form of a list of tuples—one for each key-value pair:

```
>>> eng2sp.items()
[('one','uno'), ('three', 'tres'), ('two', 'dos')]
```

The syntax provides useful type information. The square brackets indicate that this is a list. The parentheses indicate that the elements of the list are tuples.

The syntax provides useful type information. The square brackets indicate that it's a list. The parenthesis indicate that the elements in the list are tuples.

Aliasing and copying

Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

If you want to modify a dictionary and keep a copy of the original, use the `copy` method. For example, `opposites` is a dictionary that contains pairs of opposites:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

`alias` and `opposites` refer to the same object; `copy` refers to a fresh copy of the same dictionary. If we modify `alias`, `opposites` is also changed:

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

If we modify `copy`, `opposites` is unchanged:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

Aliasing and copy using function- copy()

```
>>> d={1 : "I am Python",
...    2: "I Love Python",
...    3: "I can do code quickly"
...    }
>>> d1  =  d
>>>
>>> d_c = d.copy()
>>>
>>> d[1] = "INT213"
>>>
>>> d
{1: 'INT213', 2: 'I Love Python', 3: 'I can do code quickly'}
>>>
>>> d1
{1: 'INT213', 2: 'I Love Python', 3: 'I can do code quickly'}
>>>
>>> d_c
{1: 'I am Python', 2: 'I Love Python', 3: 'I can do code quickly'}
>>>
```

Sparse matrices

consider a sparse matrix like this

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

The list representation contains a lot of zeroes:

```
matrix = [ [0,0,0,1,0],
           [0,0,0,0,0],
           [0,2,0,0,0],
           [0,0,0,0,0],
           [0,0,0,3,0] ]
```

An alternative is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix:

```
matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}
```

We only need three key-value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.

To access an element of the matrix, we could use the [] operator:

```
matrix[0,3]
1
```

Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.

There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key:

```
>>> matrix[1,3]
KeyError: (1, 3)
```

The `get` method solves this problem:

```
>>> matrix.get((0,3), 0)
1
```

The first argument is the key; the second argument is the value `get` should return if the key is not in the dictionary:

```
>>> matrix.get((1,3), 0)
0
```

QUIZ

Which of the following statements create a dictionary?

a) d = {}
b) d = {“john”:40, “peter”:45}
c) d = {40:”john”, 45:”peter”}
d) All of the mentioned

Items are accessed by their position in a dictionary and All the keys in a dictionary must be of the same type.

a. True

b. False

Dictionary keys must be immutable

a. True

b. False

In Python, Dictionaries are immutable

a. True

b. False

What will be the output of the following Python code snippet?

```
a={1:"A",2:"B",3:"C"}
for i,j in a.items():
    print(i,j,end=" ")
```

a) 1 A 2 B 3 C
b) 1 2 3
c) A B C
d) 1:”A” 2:”B” 3:”C”

Select the correct way to print Emma's age.

```
student = {1: {'name': 'Emma', 'age': '27', 'sex': 'Female'},
           2: {'name': 'Mike', 'age': '22', 'sex': 'Male'}}
```

a. student[0][1]

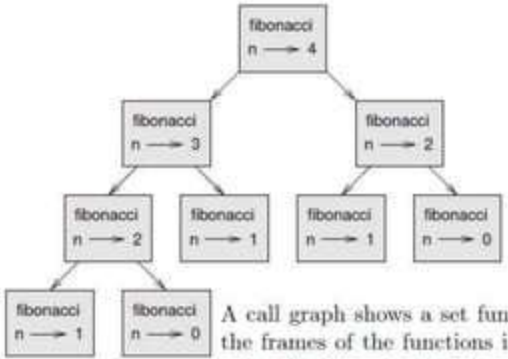
b. student[1]['age']

c. student[0]['age']

d. student[2]['age']

the `fibonacci` function, you might have noticed that the bigger the argument you provide, the longer the function takes to run. Furthermore, the run time increases very quickly.

To understand why, consider this **call graph** for `fibonacci` with `n=4`:



A call graph shows a set function frames, with lines connecting each frame to the frames of the functions it calls. At the top of the graph, `fibonacci` with `n=4` calls `fibonacci` with `n=3` and `n=2`. In turn, `fibonacci` with `n=3` calls `fibonacci` with `n=2` and `n=1`. And so on.

Count how many times `fibonacci(0)` and `fibonacci(1)` are called. This is an inefficient solution to the problem, and it gets far worse as the argument gets bigger.

A good solution is to keep track of values that have already been computed by storing them in a dictionary. A previously computed value that is stored for later use is called a **hint**. Here is an implementation of `fibonacci` using hints:

```
previous = {0:1, 1:1}

def fibonacci(n):
  if previous.has_key(n):
    return previous[n]
  else:
    newValue = fibonacci(n-1) + fibonacci(n-2)
    previous[n] = newValue
    return newValue
```

Using this version of `fibonacci`, our machines can compute `fibonacci(40)` in an eyeblink. But when we try to compute `fibonacci(50)`, we see the following:

```
>>> fibonacci(50)
20365011074L
```

The `L` at the end of the result indicates that the answer +(20,365,011,074) is too big to fit into a Python integer. Python has automatically converted the result to a long integer.

Fibonacci using Dictionary

```
fib={0:0,1:1}
def fibo(n):
    if(n==0):
      return 0
    if(n==1):
      return 1
    else:
        for i in range(2, n+1):
            fib[i]=fib[i-1]+fib[i-2]
        return(fib)
print(fibo(5))
```

Counting letters

Use get()
The get() method returns the value of the item with the specified key.
Syntax:
dictionary.get(*keyname*, *value*)

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.get("model")

print(x)
```

```
>>> d = { }
>>> for i in "Hi-Hi-Hi-Hi-Hi":
...     d [i] = d . get (i , 0) +1
...
...
>>> d
{'H': 5, 'i': 5, '-': 4}
>>>
```

Counting letters

Dictionaries provide an elegant way to generate a histogram:

```
>>> letterCounts = {}
>>> for letter in "Mississippi":
...    letterCounts[letter] = letterCounts.get (letter, 0) + 1
...
>>> letterCounts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

We start with an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.

It might be more appealing to display the histogram in alphabetical order. We can do that with the `items` and `sort` methods:

```
>>> letterItems = letterCounts.items()
>>> letterItems.sort()
>>> print letterItems
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Iterate over Python dictionaries using for loops

```
d={'red':1,'green':2,'blue':3}
for color_key, value in d.items():
        print(color_key,'corresponds to', d[color_key])
```

OUTPUT:

```
blue corresponds to 3
green corresponds to 2
red corresponds to 1
```

Remove a key from a Python dictionary

```
myDict = {'a':1,'b':2,'c':3,'d':4}
print(myDict)
if 'a' in myDict:
   del myDict['a']
print(myDict)
```

OUTPUT:

```
{'d': 4, 'a': 1, 'b': 2, 'c': 3}
 {'d': 4, 'b': 2, 'c': 3}
```

Sort a Python dictionary by key

```
color_dict = {'red':'#FF0000',
              'green':'#008000',
               'black':'#000000',
              'white':'#FFFFFF'
                }

for i in sorted(color_dict):
   print(key," :", color_dict[ i ]))
```

OUTPUT:

```
black: #000000
green: #008000
red: #FF0000
white: #FFFFFF
```

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Remove all elements from the car

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.clear()

print(car)
```

The fromkeys() method returns a dictionary with the specified keys and the specified va

Create a dictionary with 3 keys, all with the value 0:

```
x = ('key1', 'key2', 'key3')
y = 0

thisdict = dict.fromkeys(x, y)

print(thisdict)
```

Copy the car dictionary:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

x = car.copy()

print(x)
```

```
car = {
  "brand": "SWIFT",
  "model": "VDI",
  "year": 2015
}

x = car.setdefault("model", "ZDI")

print(x)
o/p:VDI
#ZDI will get printed if model item doesnt exist
```

- Use update() on dictionary

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.update({"color1": "White"})
#as color1 named key not in dictionary. so it is inserted as Last item
print(car)

car.update({"brand": "SWIFT"})

print(car)
```

- **The pop()** method removes the specified item from the dictionary.
- The **popitem()** method removes the item that was last inserted into the dictionary. In versions before 3.7, the popitem() method removes a random item.

Remove "model" from the dictionary:

```
car = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

car.pop("model")

print(car)
```

```
car = {
  "model": "Mustang",
  "year": 1964,
"brand": "Ford",
}

car.popitem()

print(car)
```

Find the maximum and minimum value of a Python dictionary

```
>>> d={1:3000,
...    2:45,
...    3:12,
...    4:1000,
...    5:-2
...    }
>>> d.values()
    dict_values([3000, 45, 12, 1000, -2])
>>> max(d)
    5
>>> min(d)
    1
>>> max(d.values())
    3000
>>> min(d.values())
    -2
>>> sorted(d)
    [1, 2, 3, 4, 5]
>>> sorted(d.items())
    [(1, 3000), (2, 45), (3, 12), (4, 1000), (5, -2)]
>>> sorted(d.items())
    [(1, 3000), (2, 45), (3, 12), (4, 1000), (5, -2)]
>>> sorted(d.values())
    [-2, 12, 45, 1000, 3000]
```

Concatenate two Python dictionaries into a new one

```
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
dic4 = {}
for d in (dic1, dic2, dic3):
        dic4.update(d)
print(dic4)
```

OUTPUT:

{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}

Test whether a Python dictionary contains a specific key

```
fruits = {}
fruits["apple"] = 1
fruits["mango"] = 2
fruits["banana"] = 4

if "mango" in fruits:
    print("Has mango")
else:
    print("No mango")

if "orange" in fruits:
    print("Has orange")
else:
    print("No orange")
```

OUTPUT:

```
Has mango
 No orange
```

QUESTIONS

Q1:Write a Python script to add a key to a dictionary.

Sample Dictionary : {0: 10, 1: 20}
Expected Result : {0: 10, 1: 20, 2: 30}

Q2: Write a Python script to concatenate following dictionaries to create a new one.
Sample Dictionary :
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
Expected Result : {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}

Q3: Write a Python script to check if a given key already exists in a dictionary.

Q4: Write a Python script to print a dictionary where the keys are numbers between 1 and 15 (both included) and the values are square of keys.

Sample Dictionary
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225}

- #Add two dictionaries and get results.
- #concatenate dictionaries
- d1={'A':1000,'B':2000}
- d2={'C':3000}
- d1.update(d2)
- print("Concatenated dictionary is:")
- print(d1)
- o/p: Concatenated dictionary is:
- {'A': 1000, 'B': 2000, 'C': 3000}

```
#concatenate dictionaries
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
dic4 = {}
for i in (dic1, dic2, dic3):
   dic4.update(i)
print(dic4)

o/p:
{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
```

```
#3: Write a Python script to check if a given key already
    exists in a dictionary.
d = {1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}
def present(i):
 if i in d:
    print(i ,' key is present in the dictionary')
 else:
    print(i, 'key is not present in the dictionary')
present(20)
present(3)
```

o/p:

20 key is not present in the dictionary

3 key is present in the dictionary

```
##sum and multiplication of dictionary
   elements
d = {1: 10, 2: 20, 3: 1, 4: 1, 5: 1, 6: 1}
print(sum(d.values()))
mul=1
for i in d:
   mul = mul * d[i]
print("Multiplication is", mul)
```

Use of __setitem__ ( )

- myDict = {1:100, 2:200}
- myDict.__setitem__(33,300)
- myDict
- {1: 100, 2: 200, 33: 300}

Add user input() in a dictionary

- d = { }
- for i in range(3):
  - i = input("enter Key")
  - j = input("enter value")
  - d[i] = j

print(d)

Take dictionary items() from user

```
employees = {}

for i in range(3):
    name = input("Enter employee's name: ")
    salary = input("Enter employee's salary: ")

    employees[name] = salary


print(employees)
```

```
Enter employee's name: 1
Enter employee's salary: 444444444444
Enter employee's name: 2
Enter employee's salary: 4444444444
Enter employee's name: 5
Enter employee's salary: 666666666666666666666
{'1': '444444444444', '2': '4444444444', '5': '666666666666666666666666'}
>>>
```

Q6: Write a Python script to merge two Python dictionaries.

Q7: Write a Python program to sum all the items in a dictionary.

Q8: Write a Python program to multiply all the items in a dictionary.

Q9: Write a Python program to remove a key from a dictionary.

Q10: Write a Python program to remove duplicates from Dictionary.

Q11: Write a Python program to create and display all combinations of letters, selecting each letter from a different key in a dictionary.

Sample data : {'1':['a','b'], '2':['c','d']}
Expected Output:
ac
ad
bc
bd

Q9: Write a Python program to remove a key from a dictionary.

- d = {'a':100, 'b':200, 'c':300, 'd':400}
- del d['a']
- >>>d
- {'b': 200, 'c': 300, 'd': 400}

File Edit Format Run Options Window Help

```
student =       {'id1': {'name': ['Aman'],    'class': ['2nd year'],    'subject_integration': ['CSE']   },
                 'id2': {'name': ['Aman'],    'class': ['2nd year'],    'subject_integration': ['CSE']   },
                  'id3':{'name': ['Ram'],     'class': ['2nd year'],    'subject_integration': ['CSE']   },
                 'id4': {'name': ['Ram'],     'class': ['2nd year'],    'subject_integration': ['CSE']   },
                }

result = {}

for key,value in student.items():
    if value not in result.values():
        result[key] = value

print(result)
```

```
========== RESTART: C:/Users/Asus/AppData/Local/Programs/Python/Python310/11.py ==========
{'id1': {'name': ['Aman'], 'class': ['2nd year'], 'subject_integration': ['CSE']}, 'id3': {'name': ['Ram'], 'class': ['2nd year'], 'subject_integration': ['CSE']}}
>>>
```

```
# Use function and print players details of  a dictionary
def indian_cricket(d):
    for i in d:
        print("Details of Players are", d[i])
ind = {'test1':{'Dhoni':75, 'Kohli':170 }, 'test2':{'Dhoni':30, 'Pujara': 45} }
indian_cricket(ind)
```

o/p

Details of Players are {'Dhoni': 75, 'Kohli': 170}

Details of Players are {'Dhoni': 30, 'Pujara': 45}

- Define a python function 'indian_cricket(d)' which reads a dictionary of the following form and identifies the player with the highest total score. The function should **return a pair (, topscore),** where playername is the name of the player with the **highest score** and **topscore** is the total runs scored by the player.

Input is:

```
indian_cricket (
{'test1':{'Dhoni':75, 'Kohli':170},
'test2':{Dhoni': 30, 'Pujara': 45}
})
```

Solution:

```
maxdic={}
dic={'test1':{'Dhoni':75, 'Kohli':170 },
     'test2':{'Dhoni':30, 'Pujara': 45}
    }

for keymain,valuemain in dic.items():  #keymain is test1 , test ... keys
    max=0
    key1=''
    dicin={}
    for key,value in valuemain.items():
        if(value>max):
            max=value
            key1=key
    dicin[key1]=max              #key1 is used for Dhoni, Kohli, Pujara
    maxdic[keymain]=dicin

print("Topscore Player wise in different Tests is", maxdic)
```

```
Topscore Player wise in different Tests is {'test1': {'Kohli': 170}, 'test2': {'Pujara': 45}}
>>
```

```
maxdic={}
dic={'test1':{'Dhoni':75, 'Kohli':170 },
    'test2':{'Dhoni':30, 'Pujara': 45}
    }
for keymain,valuemain in dic.items():  #keymain is test1 , test ... keys
    max=0
    key1=''
    dicin={}
    for key,value in valuemain.items():
        if(value>max):
            max=value
            key1=key
    dicin[key1]=max          #key1 is used for Dhoni, kohli, Pujara
    maxdic[keymain]=dicin
print("Topscore Player wise in different Tests is", maxdic)
```